서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

Programming Assignment Report - #1

Yonwoo Choi (2022-28614)
yhugestar@snu.ac.kr
Seoul National University - Advanced Graphics (Spring 2023)

Abstract

In this programming assignment, I have implemented a 3D crowd simulation called "Boids", an iconic agent-based model that mimics the collective behavior of flocks of birds or schools of fishes. The model was originally proposed by Craig Reynolds, and I follwed the provided pseudocode and guidelines to implement the simulation. The primary objective was to recreate complex emergent behaviors by defining simple rules for each agent.

The assignment consisted of three steps. In step 1, I have implemented basic behaviors such as separation, alignment, cohesion, and boundary, which are essential for agents to interact and move in a coordinated manner within the simulation. In step 2, I have added advanced behaviors, including goal seeking, predator avoidance, and obstacle avoidance, to create a more realistic and dynamic environment for the agents. Finally in the optional step 3, we utilized a KD-tree data structure to improve the efficiency of the nearest neighbor search and enable large-scale simuations. This step involved a time comparison analysis to demonstrate the benefits of using KD-tree in the simulation.

1 Step 1

1.1 Separation

PseudoCode

```
for each boid $i$ in the flock do
    separation ← zero vector;
    count ← 0;
    for each boid $j$ in the flock do
        if $i \neq j$ and $distance(i, j) < separationDistance$ then
            separation ← separation - $(j.position - i.position)$; count ← count + 1;
        end
    end
    if count > 0 then
        separation ← separation / count;
        $i.velocity \leftarrow i.velocity + separation * separationWeight$;
    end
end
```

Code Listing

Listing 1: Separation Function

```python
def separation(self, boid, neighbors, separation_radius=2.0):
    separation_force = np.zeros(3)
    for neighbor in neighbors:
        if np.linalg.norm(boid.position - neighbor.position) < ↪
          separation_radius:
            separation_force += (boid.position - neighbor.position↪
          )
    return separation_force * self.separation_weight
```

Explanation

The separation property's main purpose is to maintain a certain distance between individual boids to prevent crowding and collisions. This is achieved by calculating a repulsive force that pushes each boid away from others that are too close. To determine which boids are considered neighbors for the separation property, a predefined distance threshold is used. Any boid within this distance is considered a neighbor and will have an impact on the boid's separation behavior. For each boid, the algorithm calculates the separation force by considering all its neighbors within the threshold distance.[1] The force is a vector that points away from each neighbor, and its magnitude is inversely proportional to the distance between the boid and its neighbor. In other words, the closer a neighbor is, the stronger the repulsive force. The separation forces from all neighbors are summed up to obtain the total separation force acting on the boid. This ensures that the boid is pushed away from all its close neighbors at once. The total separation force is typically normalized to maintain a constant force magnitude, which ensures smoother motion and prevents sudden, jerky movements. It can also be scaled by a user-defined parameter to adjust the strength of the separation behavior. The calculated separation force is combined with the other two forces (alignment and cohesion) to determine the overall motion of the boid. The boid's velocity is updated according to the combined force, which in turn affects its position in the simulation.[2]

1.2 Alignment

PseudoCode

```
for each boid $i$ in the flock do
    alignment ← zero vector;
    count ← 0;
    for each boid $j$ in the flock do
        if $i \neq j$ and $distance(i, j) < alignmentDistance$ then
            alignment ← alignment + j.velocity;  count ← count + 1;
        end
    end
    if count > 0 then
        alignment ← alignment / count
        i.velocity ← i.velocity + alignment * alignmentWeight;
    end
end
```

Code Listing

Listing 2: Alignment Function

```
def alignment(self, boid, neighbors, alignment_radius=5.0):
    average_velocity = np.zeros(3)
    count = 0
    for neighbor in neighbors:
        distance = np.linalg.norm(boid.position − neighbor.position)
        if distance < alignment_radius and distance > 0:
            average_velocity += neighbor.velocity
            count += 1
    if count > 0:
        average_velocity /= count
        return (average_velocity − boid.velocity) ∗ self.alignment_weight
    else:
        return np.zeros(3)
```

Explanation The primary purpose of the alignment property is to make each boid align its direction with the average direction of its neighboring boids, which contribute to the smooth, coordinated movement of the entire flock. The primary objective of the alignment property is to ensure that each boid moves in a similar direction as its neighbors. This leads to the coordinated movement of the flock, as seen in real-life flocking behavior. To determine which boids are considered neighbors for the alignment property, a predefined distance threshold is used.[1] Any boid within this distance is considered a neighbor and will have an impact on the boid's alignment behavior. For each boid, the algorithm calculates the average direction of its neighbors by summing up the velocity vectors of all the neighboring boids and dividing the result by the number of neighbors. This gives an overall direction that represents the average movement of the local flock. The alignment force is calculated as the difference between the average direction of the neighbors and the current direction of the boid. This force represents the required change indirection for the boid to align with its neighbors. It can also be scaled by a user-defined parameter to adjust the strength of the alignment behavior.

1.3 Cohesion

PseudoCode

```
for each boid $i$ in the flock do
    cohesion ← zero vector;
    count ← 0;
    for each boid $j$ in the flock do
        if $i \neq j$ and $distance(i, j) < cohesionDistance$ then
            cohesion ← cohesion + j.position;  count ← count + 1;
        end
    end
    if count > 0 then
        cohesion ← cohesion / count;  direction ← cohesion - i.position;  direction ← normalize(direction);  i.velocity ← i.velocity + direction * cohesionWeight;
    end
end
```

Code Listing

Listing 3: Cohesion Function

```
def cohesion(self, boid, neighbors, cohesion_radius=5.0):
    center_of_mass = np.zeros(3)
    count = 0
    for neighbor in neighbors:
        distance = np.linalg.norm(boid.position − neighbor.position)
        if distance < cohesion_radius and distance > 0:
            center_of_mass += neighbor.position
            count += 1
    if count > 0:
        center_of_mass /= count
        return (center_of_mass − boid.position) ∗ self.cohesion_weight
    else:
        return np.zeros(3)
```

Explanation

The primary purpose of the cohesion property is to make each boid move towards the average position of its neighboring boids, resulting in the formation and maintenance of a cohesive flock. The primary objective of the cohesion property is to ensure that each boid stays close to its neighbors, forming a cohesive group that moves together as a single unit.[1] To determine which boids are considered neighbors for the cohesion property, a predefined distance threshold is used. Any boid within this distance is considered a neighbor and will have an impact on the boid's cohesion behavior. For each boid, the algorithm calculates the average position of its neighbors by summing up the position vectors of all the neighboring boids and dividing the result by the number of neighbors. This gives an overall position that represents the center of mass of the local flock. The cohesion force is calculated as the difference between the average position of the neighbors and the current position of the boid. This force represents the required change in position for the boid to move closer to the center of the local flock. Overall, the cohesion property enables the boids to maintain a close-knit formation that resembles the group dynamics observed in real-life flocks.

1.4 Boundary

PseudoCode

```
for each boid i in the flock do
    if i.position.x < minX then
        i.velocity.x ← i.velocity.x + boundaryWeight
    end
    if i.position.x > maxX then
        i.velocity.x ← i.velocity.x - boundaryWeight
    end
    if i.position.y < minY then
        i.velocity.y ← i.velocity.y + boundaryWeight
    end
    if i.position.y > maxY then
        i.velocity.y ← i.velocity.y - boundaryWeight
    end
    if i.position.z < minZ then
        i.velocity.z ← i.velocity.z + boundaryWeight
    end
    if i.position.z > maxZ then
        i.velocity.z ← i.velocity.z - boundaryWeight
    end
end
```

Code Listing

Listing 4: Boundary Function

```python
def boundary(self, boid):
    boundary_force = np.zeros(3)
    for i in range(3):
        if boid.position[i] < self.bounds[i * 2]:
            boundary_force[i] = self.bounds[i * 2] − boid.position[i]
        elif boid.position[i] > self.bounds[i * 2 + 1]:
            boundary_force[i] = self.bounds[i * 2 + 1] − boid.↩
  position[i]
    return boundary_force * self.boundary_weight
```

Explanation

The basic idea of the boundary behavior is to prevent agents from flying out of the simulation space. In this implementation, the simulation space is considered to be a cube with a given size. If a boid reaches the edge of the cube, its velocity is adjusted to steer it back inside the simulation space. This implementation uses a simple approach where a boid's position is checked against the boundary limits in each dimension and if the boid is beyond any of the limits, its velocity is adjusted to steer it back into the cube. The adjustment is based on the distance between the boid's current position and the boundary limit in that dimension, with the velocity component in that direction multiplied by a boundary weight factor. The implementation is done using two for loops, where the outer loop iterates over each boid in the flock, and the inner loop iterates over the x,y,z dimensions to check if the boid is outside the boundary limits. If it is outside the limits, the velocity is ajdusted as described above.

2 Step 2

2.1 Goal Seeking

PseudoCode

```
for each boid $i$ in the flock do
    goalForce ← zero vector
    vectorToGoal ← goalPosition - i.position
    normalize(vectorToGoal)
    goalForce ← goalWeight * vectorToGoal
    i.velocity ← i.velocity + goalForce
end
```

Code Listing

Listing 5: Goal Seeking

```python
def goal_seeking(self, boid):
    vector_to_goal = self.goal_position − boid.position
    normalized_vector = vector_to_goal / np.linalg.norm(↩
      vector_to_goal)
    return self.weights['goal'] * normalized_vector
```

Explanation

The goal seeking function is responsible for steering the boid towards a predefined goal position. The idea is to calculate a force that directs the boid towards the goal. The steps are as follows:

1. For each boid in the flock, initialize a zero vector called **goalForce**.
2. Calculate the **vectorToGoal** by subtracting the boid's position from the goal position.
3. Normalize the **vectorToGoal** to have a unit length. This ensures that the direction towards the goal is considered, without affecting the speed of the boid.
4. Multiply the **vectorToGoal** by the **goalWeight** to control the influence of the goal-seeking behavior on the boid's overall movement. This weighted force is now the **goalForce**.
5. Update the boid's velocity by adding the **goalForce**.

2.2 Predators

PseudoCode

```
for each boid $i$ in the flock do
    avoidanceForce ← zero vector
    for each predator $p$ in predatorPositions do
        distanceToPredator ← distance(i.position,
          p.position)
        if distanceToPredator < predatorRadius then
            escapeVector ← i.position - p.position
            normalize(escapeVector)
            avoidanceForce ← avoidanceForce +
              predatorWeight * escapeVector
        end
    end
    i.velocity ← i.velocity + avoidanceForce
end
```

Code Listing

Listing 6: Predators

```python
def predator_avoidance(self, boid, radius=5.0):
    avoidance_force = np.zeros(3)

    for predator in self.predator_positions:
        distance_to_predator = np.linalg.norm(predator - boid.position)

        if distance_to_predator < radius:
            escape_vector = boid.position - predator
            normalized_vector = escape_vector / distance_to_predator
            avoidance_force += self.weights['predator'] * normalized_vector

    return avoidance_force
```

Explanation

The predator avoidance function helps boids steer away from predators. The main idea is to calculate a repulsive force that pushes boids away from the predator. The steps are as follows:

1. For each boid in the flock, initialize a zero vector called **avoidanceForce**.
2. For each predator in the simulation, calculate the **distanceToPredator** between the boid and the predator.
3. If **distanceToPredator** is less than a predefined avoidance radius, calculate the **escapeVector** by subtracting the predator's position from the boid's position.
4. Normalize the **escapeVector** to have a unit length.
5. Multiply the **escapeVector** by the **predatorWeight** to control the influence of predator avoidance on the boid's overall movement. Add the weighted force to the **avoidanceForce**.
6. Update the boid's velocity by adding the **avoidanceForce**.

2.3 Obstacle Avoidance

PseudoCode

```
for each boid $i$ in the flock do
    avoidanceForce ← zero vector
    for each obstacle $o$ in obstacles do
        distanceToObstacle ← distance(i.position,
          o.position)
        if distanceToObstacle < obstacleRadius then
            escapeVector ← i.position - o.position
            normalize(escapeVector)
            avoidanceForce ← avoidanceForce +
              obstacleWeight * escapeVector
        end
    end
    i.velocity ← i.velocity + avoidanceForce
end
```

Code Listing

Listing 7: Obstacle Avoidance

```python
def obstacle_avoidance(self, boid, radius=3.0):
    avoidance_force = np.zeros(3)

    for obstacle in self.obstacles:
        distance_to_obstacle = np.linalg.norm(obstacle.position - boid.position)

        if distance_to_obstacle < radius:
            escape_vector = boid.position - obstacle.position
            normalized_vector = escape_vector / distance_to_obstacle
            avoidance_force += self.weights['obstacle'] * normalized_vector

    return avoidance_force
```

Explanation

The obstacle avoidance function is responsible for preventing the boid from colliding with obstacles in the environment. The main idea is to calculate a force that repels the boid from the obstacles. The steps are as follows:

1. For each boid in the flock, initialize a zero vector called **avoidanceForce**.
2. Iterate through all the obstacles in the environment.
3. Calculate the distance between the boid and the obstacle.
4. If the distance is less than a predefined radius, calculate the **escapeVector** by subtracting the obstacle's position from the boid's position.
5. Normalize the **escapeVector** and multiply it by the **obstacleWeight** to get the weighted avoidance force for the current obstacle.
6. Add this weighted force to the **avoidanceForce**.
7. Update the boid's velocity by adding the **avoidanceForce**.

3 Step 3

3.1 KD-Tree for Large-scale Simulation

Overview

KD-trees, short for *k*-dimensional trees, are a versatile data structure used for organizing points in a *k*-dimensional space. They are particularly useful for nearest neighbor searches, range queries, and other spatial search problems. KD-trees recursively partition the space into hyperrectangles, allowing for efficient search and storage of data points. The tree structure is composed of nodes that store a single data point and separate the space into two distinct regions along a single dimension. The search and insert operations have an average time complexity of $O(\log n)$, where n is the number of data points, making KD-trees an efficient choice for high-dimensional data analysis. [3]

From the **scipy** library, I've imported the **KDTree** data structure for large-scale simulation speedup. I've added a KD tree data structure to the **update** function of the **BoidSimulation** class to store neighbors. I've set the radius value to a specific value. In general, a larger radius will result in smoother and more natural looking flocking behavior, but it will also require more computations and can become computationally expensive for very large simulations. I have set the radius value based on the size of the simulation space, $\frac{1}{3}$ of the diagonal length of the simulation space.

Code Listing

Listing 8: Using KD Tree in Nearest Neighbor Search

```
def update(self):
    boid_positions = np.array([boid.position for boid in self.boids])
    boid_kdtree = KDTree(boid_positions)

    for boid in self.boids:
        if self.use_kdtree:
            neighbor_indices = boid_kdtree.query_ball_point(boid.position, self.neighbor_radius)
            neighbors = [self.boids[i] for i in neighbor_indices if self.boids[i] is not boid]

        else:
            neighbors = [other for other in self.boids if other is not boid]
```

Listing 9: Testing on different population sizes

```
population_sizes = [10, 50, 100, 200, 400, 800]
neighbor_radius_values = [5]

for pop_size in population_sizes:
    for neighbor_radius in neighbor_radius_values:
        simulation_no_kdtree = BoidSimulation(pop_size, bounds, use_kdtree=False, neighbor_radius=neighbor_radius)
        simulation_with_kdtree = BoidSimulation(pop_size, bounds, use_kdtree=True, neighbor_radius=neighbor_radius)

        time_no_kdtree = measure_time(simulation_no_kdtree, 100)
        time_with_kdtree = measure_time(simulation_with_kdtree, 100)

        print(f"Population: {pop_size}, Neighbor radius: {neighbor_radius}")
        print(f"Execution time without KD Tree: {time_no_kdtree:.2f} seconds")
        print(f"Execution time with KD Tree: {time_with_kdtree:.2f} seconds")
```

3.2 Time comparison

For comparison, I have set the neighbor radius value to 5 and ran tests on different population sizes. The KD-Tree execution time is compared to the list data structure running time in the nearest neighbor search.

Population	With List	With KD-Tree
10	0.13 s	0.05 s
50	2.66 s	0.30 s
100	10.49 s	0.81 s
200	41.53 s	2.20 s
400	164.31 s	4.66 s
800	650.37 s	12.35 s

Table 1: Execution time comparison for Boid simulation with and without KD-Tree

The asymptotic running time of the Boid simulation without using KD-Tree is $O(n^2)$, where n is the population size. By incorporating KD-Tree, the running time is significantly reduced, approaching $O(N)$ sub-linear time complexity.

References

[1] R. Strauss, "Boids: A simple implementation of the boids flocking algorithm," 2021. Accessed: 2023-04-14.

[2] C. W. Reynolds, "Boids (flocks, herds, and schools: a distributed behavioral model)," 1987. Accessed: 2023-04-18.

[3] Wikipedia contributors, "K-d tree — Wikipedia, The Free Encyclopedia," 2021. Accessed: 2023-04-17.